Chapter THREE
Inner Classes

Exam Objectives

Create inner classes including static inner class, local class, nested class, and anonymous inner class.

Classes

In Java we have classes:

```
class Computer {

}
```

Classes have two types of members, attributes (or fields) and methods (or functions):

```
class Computer {
     int serialNumber;
     void executeCommand() {
         // Do something
     }
}
```

This way, our programs are a collection of classes.

```
class Computer {
     // Here goes the definition of the class
}
class Desktop {
     // Here goes the definition of the class
}
class Printer {
     // Here goes the definition of the class
}
class Programmer {
     // Here goes the definition of the class
}
```

Inner classes

Java gives us flexibility in the way we can design our classes.

For this, there's a third type of member a class can have, an **INNER CLASS**.

```
class Computer {
     String serialNumber;
     void executeCommand() { }
     class Processor {
         // Here goes the definition of the class
     }
}
```

Inner classes are also known as nested classes. In theory, you could have many levels of classes.

I have a hard time thinking about what would be the benefit of having more than one level of inner classes.

```
class LevelOne {
     class LevelTwo {
         class LevelThree {
             class LevelFour {
                 /** Finally do something */
             }
         }
     }
}
```

Another thing. We always talk about inner **CLASSES**, but actually, we can have inner **ABSTRACT CLASSES**, inner **INTERFACES**, and inner **ENUMS**.

```
class Computer {
     abstract class Processor { }
     interface Pluggable { }
     enum PORTS {
         USB2, USB3, ESATA, HDMI
     }
}
```

But we are going to focus on simple inner classes. There are four types of them:

- **STATIC** inner classes
- **NON-STATIC** inner classes
- **LOCAL** classes
- **ANONYMOUS** classes

Defining a static inner class

```
class Computer {
     static class Mouse {
     }
}
```

Using a static inner class

```
Computer.Mouse m = new Computer.Mouse();
```

Static inner classes are accessed through their enclosing class

---

Static classes are **INDEPENDENT** of their enclosing class. They are like ordinary classes, only that they just happen to be inside another class.

In fact, you can think of the enclosing class as a kind of a package. You can import the name of the enclosing class and use the static inner class like a normal class. Just remember that the inner static class must be a `public` member so that it can be accessed from another package.

```
import com.example.Computer.*;
public class Test {
      Mouse m = new Mouse();
     /** Rest of the definition */
}
```

And they can also be marked as `private`, `protected` or without a modifier, so they are accessible only in the package (default accessibility).

```
public class Computer {
     private static class Component { }
     protected static class MotherBoard { }
     static class Slot { }
}
```

Of course, by being a member of a class, the static inner class have access to the other members of the enclosing class, but only if they are **STATIC**.

```
public class Computer {
     private static String serialNumber = "1234X";
     public static class Mouse {
         void printSN() {
             System.out.println("MOUSE-" + serialNumber);
         }
     }
}
```

Why?

Think about it, if the static class is independent of its enclosing class, it doesn't need an instance of this, so only the static members could be used because they are associated with the class, not to a particular instance.

For that reason, a static inner class is often used as a utility class that contains common methods shared by all the objects of a class.

If you use the static inner class inside the class that defines it, you can use it in any method, block or constructor, no matter if it's static or not, since the inner class is not tied to a particular instance.

Defining a non-static inner class

```
class Computer {
     class HardDrive {
     }
}
```

Using a non-static inner class

NON-Static inner classes are accessed through an instance of their enclosing class

```
Computer c = new Computer();
Computer.HardDrive hd = c.new HardDrive();
```

Check out the type of the inner class and how the new operator is used

---

Non-static inner classes are just called inner classes.

Instances of an inner class only exist **WITHIN** an instance of the enclosing class. It's the same that when you want to use a method of a class, you **FIRST** need an instance of that class.

Once you have an instance of the enclosing class, you use the `new` operator in a (weird) different way than you typically use it.

```
Computer computer = new Computer();
Computer.HardDrive hardDrive = computer.new HardDrive();
```

You can also use the import trick to writing less, but you still need to create the inner class as always.

```
import com.example.Computer.*;
public class Test {
         Computer computer = new Computer();
         HardDrive hd = computer.new HardDrive();
         /** Rest of the definition */
}
```

Another way to get an instance of an inner class is to use a method of the enclosing class to create it, avoiding that weird syntax.

```
public class Computer {
     class HardDrive { }
     public HardDrive getHardDrive() {
         return new HardDrive();
     }
}
```

By being a member of a class, the inner class has access to the other members of the enclosing class, but this time, it **DOESN'T** matter if they are static or not.

```
public class Computer {
     private String brand = "XXX";
     private static String serialNumber = "1234X";
     public class HardDrive {
         void printSN() {
            System.out.println(
                brand + "-MOUSE-" + serialNumber
             );
         }
     }
}
```

Why?

Because to use the inner class, an instance of the enclosing class is required, ensuring that the non-static members exist (static members can be accessed anyway).

Inner classes can also be marked as `private`, `protected` or without a modifier, so they are accessible only in the package. But most of the time, since they depend on the enclosing class, they are marked as `private`.

```
public class Computer {
     private class Component { }
     protected class MotherBoard { }
     class Slot { }
}
```

Another rule is that inner classes **CANNOT** contain `static` members.

```
public class Computer {
     class HardDrive {
          // Compile-time error here
          static int capacity;
           // Compile-time error here
           static void printInfo() {
               // Definition goes here
          }
     }
}
```

Static code is executed during class initialization, but you cannot initialize a non-static inner class without having an instance of the enclosing class.

Because an inner class belongs to **ONE** instance of the enclosing class. Having a `static` member means it can be shared across instances because the member belongs to the class, but since we are talking about an inner class that cannot be shared by many instances of the enclosing class, that is not possible.

The only exception is when you define a `final static` attribute. The `final` keyword makes all the difference; it makes a constant expression, but it only works with **ATTRIBUTES** and when assigning an **NON-NULL** value.

```
public class Computer {
     class HardDrive {
          final static int capacity = 120; // It does compile!
          // Compile-time error here
          final static String brand = null;
          // Compile-time error here
          final static void printInfo() {
              // Definition goes here
          }
     }
}
```

Defining a local class

```
class Computer {
     void process() {
          class Processor {
          }
     }
}
```

Using a local class

Local classes can only be used inside the method or block that defines them

```
void process() {
        class Core { }
        Core core = new Core();
}
```

Local classes are local because they can only be used in the method or block where they are declared. Blocks are practically anything between curly braces.

```
void method() {
    class MethodLocalClass { }
    MethodLocalClass mlc = new MethodLocalClass();
    if ( 1 == 1 ) {
       class IfLocalClass { }
       IfLocalClass ilc = new IfLocalClass();
    }
    while ( true ) {
       class WhileLocalClass { }
       WhileLocalClass wlc = new WhileLocalClass();
    }
}
```

Also, notice where the instances of the local classes are created. The local class has to be used **BELOW** its definition. Otherwise, the compiler won't be able to find it.

Because a local inner class is not a member of a class, it **CANNOT** be declared with an access level, and it wouldn't make sense anyway since they are only accessible where they are declared. However, a local class can be declared as abstract or final (but not at the same time).

Local classes require an instance of their enclosing class so the method or block in which they are defined can be executed. For this reason, they can access the members of the enclosing class, but they cannot declare static members (only static final attributes), just like inner classes.

```
class Computer {
     private String serialNumber = "1234XX";
     void process() {
         class Processor {
             Processor() {
                 System.out.println(
                     "Processor #1 of computer " +
                      serialNumber
                 );
             }
         }
     }
}
```

If the local class is declared inside a method, it can access the variables and parameters of the method **ONLY** if they are declared final or are effectively final.

Effectively final is a term that means that a variable or parameter is not changed after it's initialized, even if its declaration does not use the `final` keyword.

Why?

Because an instance of a local class can be alive even after the method or block in which it is defined has finished its execution (for example, if a reference is saved in an object with greater scope). For this reason, the local class must keep an internal copy

of the variables it uses, and the only way to ensure that both copies always hold the same value it's by making the variable `final`.

So, the following code is valid because `taskName` is declared `final` while `n` doesn't change and is considered *effectively final*.

```
void process(int n) {
     final String taskName = "Task #1";
         class Processor {
             Processor() {
                 System.out.println(
                     "Processor " + n +
                      " processing " + taskName
                 );
         }
     }
}
```

But if we modify the value of `n` somewhere, an error will be generated.

```
void process(int n) {
     final String taskName = "Task #1";
     class Processor {
         Processor() {
             System.out.println(
                 "Processor " + n +    // Compile-time error
                 " processing " + taskName
             );
         }
     }
     n = 4;
}
```

Effectively final is only concerned with references, not objects or their content, because at the end of the day, we are referencing the same object.

```
void process(int n) {
     StringBuffer taskName = new StringBuffer("Task #1");
     class Processor {
         Processor() {
             System.out.println(
                 "Processor " + n +
                 " processing " + taskName // It does compile!
             );
         }
     }
     taskName.append("1"); // This is valid!
     //Uncommenting the following line will generate an error
     //taskName = new StringBuffer("Task #2");
 }
```

If you're still not sure about a declaration being *effectively final*, try adding the `final` modifier to it. If the program continues to behave in the same way, then the declaration is *effectively final*.

If the class is declared in a static method, static rules also apply, meaning that the local class only has access to the static members of the enclosing class.

Defining an anonymous class

The new operator is followed by the name of an interface or a class and the arguments to a constructor (or empty parentheses if it's an interface)

```
Computer comp = new Computer() {
     void process() {
         // Here goes the definition
     }
};
```

Look how it ends with a semicolon, like any other Statement in java

The body of the class implements the interface or extends the class referenced

---

An anonymous class is called that way because it doesn't have a name. However, an anonymous class expression doesn't declare a new class. It either **IMPLEMENTS** an existing interface or **EXTENDS** an existing class. So

```
new Computer() { }
```

is like writing

```
class [NO_NAME_CLASS] extends Computer { }
```

And if we're working with an interface

```
new Runnable() { }
```

is like writing

```
class [NO_NAME_CLASS] implements Runnable { }
```

Also, an anonymous class can be used in a declaration or a method call.

```
class Program {
     void start(Computer c) {
         // Definition goes here
     }
     public static void main(String args[]) {
         Program program = new Program();
         program.start(new Computer() {
             void process() { /** Redefinition goes here */ }
         });
     }
}
```

Since they don't have a name (well, actually the compiler gives them a random name when it creates the `.class` file), anonymous classes can't have **CONSTRUCTORS**. If you want to run some initializing code, you have to do it with an initializer block.

```
Computer t = new Computer() {
     {
         // Initializing code
     }
     void process() { /** Redefinition goes here */ }
};
```

Because anonymous classes are a type of local classes, they have the same rules:

- They can access the members of their enclosing class
- They cannot declare `static` members (only if they are `final static` variables)
- They can only access local variables (variables or parameters defined in a method) if they are final or *effectively final*.

But one thing you have to be careful with, is inheritance.

When you use an anonymous class (a subclass object), you're using a superclass reference. With this reference, you can use the attributes and methods declared in that type.

But what happens when you declare a new method on the anonymous class?

```
class Task {
     void doIt() {
         /** Here goes the definition */
     }
}
class Launcher {
     public static void main(String args[]) {
         Task task = new Task() {
             void redoIt() {
                 /** Here goes the definition */
             }
         };
         task.doIt(); // It's OK
         task.redoIt(); // Compile-time error!
     }
}
```

The program will fail. The reference doesn't know about the method `redoIt()` because it's not defined in the superclass.

Typically, you would cast the type to the subclass where the new method is defined.

```
SubClass object = (SubClass) objectWithSuperClassReference;
object.methodOnlyDefinedInTheSubclass();
```

But with an anonymous class, how does the cast is done?

It can't be done; the class has no name, so there is no way we can use the methods defined in the declaration of the anonymous class. We can only use the methods defined in the **SUPERCLASS** (be it an interface or class).

Using an anonymous class is mostly about style. If the class has a short body, it only implements one interface (if we're working with interfaces), it doesn't declare new members, and the syntax makes your code clearer, you should consider using it instead of a local or an inner class.

Shadowing

An important concept to take into account when working with inner classes (of any type) is what happens when a member of the inner class has the same name of a member of the enclosing class.

```
class Computer {
     private String serialNumber = "1234XXX";
     class HardDrive {
         private String serialNumber = "1234DDD";
         void printSN(String serialNumber) {
             System.out.println("SN: " + serialNumber);
         }
     }
}
```

In this case, the parameter `serialNumber` shadows the instance variable `serialNumber` of `HardDrive` that in turn, shadows the `serialNumber` of `Computer`.

As it's coded, the method `printSN()` will print its argument. A shadowed declaration needs something else to be properly referred.

We know that when an object wants to refer to itself, we need to use the keyword `this`.

So, if we use `this` inside an inner class, it will refer to the inner class itself.

If we need to reference the enclosing class, inside the inner class we can also use `this`, but in this way `NameOfTheEnclosingClass.this`.

```
class Computer {
     private String serialNumber = "1234XXX";
     class HardDrive {
         private String serialNumber = "1234DDD";
         void printSN(String serialNumber) {
             System.out.println(
                 "SN: " + serialNumber
             );
             System.out.println(
                 "HardDrive SN: " + this.serialNumber
             );
             System.out.println(
                 "Computer SN: " +
                    Computer.this.serialNumber
             );
         }
     }
}
```

Since it may cause confusion, it's better to avoid it and use descriptive variable names.

Key Points

- Inner classes are declared inside another class. There are four types of them: static, non-static, local and anonymous classes.
- Static classes are just inner classes marked with the `static` keyword. However, they behave more like a top-level class than an inner class.
- You don't need an instance of the enclosing class to instantiate a static class:

```
EnclosingClass.StaticClass sc = new EnclosingClass.StaticClass();
```

- A static class cannot access non-static members of the enclosing class since it doesn't require having an instance of the enclosing class to use it.
- A (non-static) inner class is like any other member of the enclosing class so that it can be marked with any access modifier.
- Outside the enclosing class' instance methods or blocks, to instantiate an inner class, you must first create an instance of the enclosing class and then:

```
EnclosingClass ec = new EnclosingClass();
EnclosingClass.InnerClass ic = ec.new InnerClass();
```

- A local class is defined within a method or block of the enclosing class.
- The only modifiers that apply to a local class are abstract and final (but not at the same time).
- You can only use a local class in the method or block where you define it, and only after its declaration.
- A local class can access the members of a class just like any other member of the class (static rules still apply).
- However, a local class can only access the parameters and local variables of a method if they are final or *effectively final*.
- *Effectively final* means that a variable cannot be modified after its initialization, even if it's not explicitly marked as `final`.
- Anonymous classes have no name, and they either extend an existing class or implement an interface:

```
ExistingClass ac = new ExistingClass() {
   // Definition goes here
};
```

- Anonymous classes cannot have constructors.
- Anonymous classes have the same rules as local classes regarding accessing members of the enclosing class and local variables of a method.
- The only methods you can call on an anonymous class are those defined in the reference type (the superclass or the interface), even though anonymous classes can define their own methods.

Self Test

1. Given:

```
public class Question_3_1 {
     interface ITest { // 1
         void m();
     }
     public static void main(String args[]) {
         ITest t = new ITest() { // 2
             public void m() {
                 System.out.println("m()");
             }
         }
         t.m();
     }
}
```

What is the result?
A. `m()`
B. Compilation fails on the declaration marked as `// 1`
C. Compilation fails on the declaration marked as `// 2`
D. An exception occurs at runtime

2. Given:

```
public class Question_3_2 {
     public static void main(String args[]) {
         Question_3_2 q = new Question_3_2();
         int i = 2;
         q.method(i);
         i = 4;
     }
     void method(int i) {
         class A {
             void helper() {
                 System.out.println(i);
             }
         }
         new A().helper();
     }
}
```

What is the result?
A. Compilation fails
B. 2
C. 4
D. An exception occurs at runtime

3. Given:

```
public class Question_3_3 {
     public static void main(String[] args) {
         Question_3_3 q = new Question_3_3() {
                 public int sum(int a, int b) { // 1
                         return a + b;
                 }
         };
         q.sum(2,6); // 2
     }
}
```

What is the result?
A. Compilation fails on the declaration marked as `// 1`
B. Compilation fails on the line marked as `// 2`
C. `8`
D. Nothing is printed

4. Given:

```
public class Question_3_4 {
     public static class Inner {
              private void doIt() {
                       System.out.println("doIt()");
              }
     }
     public static void main(String[] args) {
              Question_3_4.Inner i = new Inner();
              i.doIt();
     }
}
```

What is the result?
A. Compilation fails because an inner class cannot be `static`.
B. Compilation fails because the `Inner` class is instantiated incorrectly inside method `main`.
C. Compilation fails because the method `doIt` cannot be called in main because it is declared as `private`
D. The program prints `doIt()`

5. Given:

```
class A {
     class B {
          class C {
               void go() {
                    System.out.println("go!");
               }
          }
     }
}
public class Question_3_5 {
     public static void main(String[] args) {
          A a = new A();
          A.B b = a.new A.B(); // 1
          B.C c = b.new C(); // 2
          c.go(); // 3
     }
}
```

What is the result?
A. Compilation first fails on the line `// 1`
B. Compilation first fails on the line `// 2`
C. Compilation fails on the line `// 3`
D. `go!` is printed

6. Given:

```
public class Question_3_6 {
     private class A { // 1
         public int plusTwo(int n) {
             return n + 2;
         }
     }
     public static void main(String[] args) {
         Question_3_6.A a = new A(); // 2
         System.out.println(a.plusTwo(3));
     }
}
```

What is the result?
A. Compilation fails on the line `// 1`
B. Compilation fails on the line `// 2`
C. `5`
D. An exception occurs at runtime

7. Given:

```
public class Question_3_7 {
     public static void main(String[] args) {
         abstract class A { // 1
                 public void m() {
                     System.out.println("m()");
                 }
         }
         public class AA extends A { } // 2
     }
}
```

What change would make this code compile?
A. Remove the `abstract` keyword on the line `// 1`
B. Add the `public` keyword on the line `// 1`
C. Remove the `public` keyword on the line `// 2`
D. None. This code compiles correctly

8. Given:

```
public class Question_3_8 {
     int i = 2;
     interface A {
         int add();
     }
     public A create(int i) {
         return new A() {
             public int add() {
                 return i + 4;
             }
         };
     }
     public static void main(String[] args) {
         A a = new Question_3_8().create(8);
         System.out.println(a.add());
     }
}
```

What is the result?
A. 6
B. 12
C. Compilation fails
D. An exception occurs at runtime

Open answers page

---

02. Inheritance and Polymorphism 04. Interfaces